An Approach for Decentralized Reasoning on the Semantic Web[1]

Tzanetos Pomonis, Dimitrios Koutsomitropoulos, Georgia Solomou, Panagiotis Aggelopoulos, Sotiris P. Christodoulou and Andreas B. Gizas

High Performance Information Systems Laboratory, Computer Engineering and Informatics Dpt., University of Patras, Patras-Rio, Greece

pomonis@ceid.upatras.gr, kotsomit@hpclab.ceid.upatras.gr, solomou@ceid.upatras.gr, aggelopp@ceid.upatras.gr, spc@hpclab.ceid.upatras.gr, gizas@ceid.upatras.gr

Abstract

Reasoning is a key component for the success of Semantic Web applications, as it ensures the acquisition of added-value information from Semantic Web data. However, corresponding optimal algorithms implemented in state-of-the-art inference engines often demonstrate very high worst-case complexity. Thus, reasoning problems are usually hard to solve and powerful systems are needed in order for reasoners to produce timely responses. Naturally, the physical and logical distribution of reasoning tasks could ameliorate this situation; however, the remote deployment of inference engines is not always straightforward and has been, until recently, restrained by poor support for ontological languages like OWL or OWL2. In this paper we present a middleware for supporting decentralized reasoning, following a concrete architecture that is based on web engineering practices. By using this tool, we show how current reasoners can be employed in decentralized scenarios in a way that is seamless and transparent to the end user/developer. In addition, it is verified that performance can indeed be improved when reasoning tasks are delegated to more powerful systems; which also holds in comparison to local, in-memory reasoner deployment.

Keywords

Reasoning; Semantic Web; OWL Link; DIG; OWL; OWL API; Web Engineering

Introduction

The late success of Description Logics (DL) seems to owe much to their close relationship with ontology languages such as OWL and OWL 2 (Grau et al, 2008). Relevant DL-based reasoning systems, such as RACER, FaCT++ and Pellet have become popular accordingly. Nevertheless, it is the use of these systems within applications that can add an "AI flavor" to the current Web and make the Semantic Web deliver (Hendler, 2008).

Even as these tools exist and evolve, we have not yet experienced common-scale usage of the Semantic Web and the inference capabilities it inherently bears with. A possible reason can be the traditional intractability of the corresponding reasoning problems: Such problems in OWL DL are solved optimally in 2-NEXP time (Tobies, 2001), whilst for OWL 2 it is 3-NEXP (Kazakov, 2008). One cannot expect from the occasional user to bear with such long time intervals during his browsing and querying, nor is there any room for improvement, at least as far as DLs are concerned. Thus, a way must be found to alleviate the end-user machine from this task and delegate it to a more powerful, back-end system (usually a server), thus creating 3-tier, distributed applications.

Until recently, options for developing this kind of applications for the Semantic Web have been very limited. DIG have been for many years the default choice for communication with reasoners, but had limited support for OWL and OWL 2. On the other hand, a programmatic API can be used, for example the OWL API, a well-known Java programmatic abstraction for manipulating ontologies. The OWL API is widely used for developing ontology-based applications and allows communication with various inference engines, deployed however only in-memory (i.e., there is no remote communication) (Horridge et al, 2007).

This approach may have the advantage of reducing the message-passing load of the DIG protocol, but surely it is insufficient for developing truly decentralized Web applications and services for the Semantic Web. As DIG 2.0 specification that would solve the aforementioned problems had been for a long time in flux, most reasoners could not be used in developing a distributed web service for Semantic Web knowledge

[1] This is an extended version of a paper published at MAW 2010 (Koutsomitropoulos et al, 2010).

discovery that would fully support OWL DL or OWL2.

Even from a performance point of view, the physical distribution to different computational systems would rather accelerate the application as a whole, especially in the case of "hard" (very expressive) ontologies: there would be a point when reasoning times outmatch the communication overhead (recall the multiple exponential complexities, both in time and space) and the execution of the reasoning algorithms to a powerful back-end system appears tempting.

In this paper, we describe the design and implementation of a middleware for the decentralized invocation and usage of reasoning services. This middleware builds upon the OWL API and allows the transparent usage of remote reasoners, as if they were deployed locally. In this way, any OWL API compliant reasoner can be accommodated.

Various remote utilization and communication strategies are examined, including serializing Java objects, passing ontology entities' URIs only and investigating multithreading support. Our results confirm the assumptions on performance and appear to verify the need for remote reasoner access.

The rest of this paper is organized as follows: In Section 2 we set the background of our work by giving an insight to the reasoner communication protocols like DIG and OWLlink and examining related tools and implementations. Section 3 presents our architecture for decentralized Semantic Web applications, from a Web Engineering perspective. We document the architecture and implementation of our decentralized reasoning module in Section 4. The results obtained from using this approach with sample ontologies and two major reasoners are presented and discussed in Section 5. Finally sections 6 and 7 summarize our conclusions and give pointers towards future work.

The source code of our tool is available at: http://swig.hpclab.ceid.upatras.gr/OWLApp.

Background

In this section we first briefly review and comment on two important protocols that can be used for remote reasoner invocation, namely DIG and OWLlink. Then we examine state-of-the-art support of these protocols in current reasoners and other related tools. Finally, we discuss some recent efforts which, similar to our work, build upon the OWL API in order to accommodate remote reasoner deployment.

DIG and OWLlink

DIG is a common standard interface for DL reasoners[2], by defining a protocol for XML message passing over HTTP. DIG early versions have been quickly adopted by reasoning engines, ontology editors and many other Semantic Web applications. But despite its popularity, DIG has confronted with a number of problems, such as limitations in the supported language fragment (e.g., no support for datatypes, poor fit between DIG's notion of relations and OWL properties), a lack of elementary queries, and the absence of a mechanism to extend the protocol (a short report on these issues is made by Dickinson (2004)).

DIG's current version is called OWLlink (Liebig et al, 2008) and is based on the most recent OWL 2 specification for the primitives of the modeling language. So OWLlink is regarded as DIG's successor – upgraded in many levels – that provides an extensible protocol for communication with OWL reasoning systems. It supports many features of OWL 2, for example the notions of punning and structural equivalence. However, it does not support any part of OWL 2 beyond the level of axioms, like OWL imports.

The communication between OWLlink-compliant clients and servers is implemented through the exchange of request and response messages. By this process, client applications become capable to configure reasoners, to access reasoning services and to transmit ontologies. A basic functionality provided by OWLlink is the ability to either add axioms to a KB (Tell requests) or to retrieve information about a KB (Ask requests) (Table 1). Ask requests only cover very basic queries with respect to the given and inferred axioms of the KB. More complex queries, notably including conjunctive ones, remain to be specified by the OWLlink *Query Interface Extension*. The latter, however, is still in draft stage and a bit outdated, since it refers to the preceding DIG 2.0 proposal[3].

TABLE 1EXAMPLE OF OWLLINK ASK-REQUEST AND RESPONSE

REQUEST	RESPONSE
<GetSubClasses ol:kb="*a_KB*" ol:direct="*false*"> <ox:OWLClass ox:URI="*a_Class*"/> </GetSubClasses>	<SetOfClassSynsets> <ClassSynset> <ox:OWLClass ox:URI="&owl;*Something*"/> <ox:OWLClass ox:URI="*a_Class*"/> </ClassSynset> </SetOfClassSynsets>

[2] http://dl.kr.org/dig/index.html

[3] http:// www.sts.tu-harburg.de/ ~al.kaplunova/ dig-query-interface.html

Existing Support by Reasoners and Tools

Apart from those DL reasoners that support the DIG interface (e.g., CEL, FaCT++, Racer, Pellet and KAON2) ontology editors, as well as many other applications and middleware, make use of DIG for reasoning over OWL ontologies (e.g., Instance Store, OntoXpl and the Jena framework). As far as OWLlink is concerned, although still in draft stage, it is supported or it is about to be done so by widely used reasoning systems like RacerPro, Pellet and FaCT++. RacerPro had initially implemented OWLlink as part of its 1.9.3 version and kept supporting it in its current version, RacerPro 2.0. QuOnto is another reasoner which is being aligned with OWLlink, through its OBDA extension (Rodriguez and Calvanese, 2008). What is more, Protégé and OntoTrack have already committed to implementing OWLlink (Liebig et al, 2008), a fact that also holds for various other developers of ontology editing and browsing tools. Finally, an OWLlink plugin for Protégé 4.1 is already available[4].

Related Implementations

To our knowledge, methods for remote reasoner deployment are not very common in Semantic Web applications, other than proprietary designed facilities, for example the usage of the JRacer API (Koutsomitropoulos et al, 2006). Additionally, as shown in the previous section, one cannot always count on DIG as a reasoner interface and this comes at a price.

The introduction of the OWLlink API (Noppens et al, 2010) offers a reference implementation of the OWLlink protocol on top of the OWL API. By using this API, a client-server communication between an application and a compliant reasoner can be maintained, similar to our work. However, the only supported OWLlink requests are the ones that can be mapped to OWLReasoner. Thus, calls to OWLReasoner methods that do not correspond to specified OWLlink Core requests-currently limited-may not be supported.

In addition, the HERAKLES system (Bock et al, 2009) implements an approach for remote reasoner usage, based on the OWL API and Java Remote Method Invocation (RMI). An important feature of HERAKLES is the ability to delegate reasoning tasks to multiple remote reasoners in a competitive manner, which reportedly appears to improve the overall response time. The system is under development and is currently being aligned with OWLlink.

Architecture for Reasoning-based Applications

Many of the first approaches for developing Web applications were slight modifications of traditional software engineering procedures, as a Web application used to be considered nothing more than a typical software product. When Web Engineering was proposed and started to be considered as a discrete domain, rather than a sub-domain of traditional Software Engineering (Murugesan et al, 1999), the Web application development approaches turned to a more specific consideration of the Web context. This has resulted in most of today's Web application development techniques to be extensions of standard software engineering, enriched with technologies and methodologies from different domains, i.e. Hypermedia, Multimedia, Databases, Networks, Human-Computer Interaction, Distributed Computing.

A number of Web developers have adopted the term Web Engineering to describe the theory of Web application development. In Web Engineering, a balance is struck among the programming, publishing and business aspects of developing Web applications (Deshpande & Hansen, 2001). As a result, there are a lot of different Web application development approaches that fall under the domain of Web Engineering. One of the main aspects during the web engineering lifecycle is to take into consideration the application's specific characteristics and requirements, i.e. heavy data load, many user transactions, rich user interface, Web 2.0 technologies, semantic infrastructure. As explained in section 1, this is mainly the situation for Semantic Web applications, where their high complexity calls for modular as well as decentralized solutions. Figure 1 depicts exactly this kind of distributed architecture, which follows the 3-tier paradigm, this time extended for the needs of Semantic Web applications (Pomonis et al, 2009).

The respective architecture has the advantage of providing the desired physical and logical independence between the discrete Web (interface) and semantic (knowledge infrastructure) components which are handled by separate tiers. A middle tier is responsible for the interconnection functions and also handles the advanced logical operations. In addition, such an architecture can be readily helpful in supporting future extensions of the target Web application, by providing a great level of flexibility. In this sense, it can enable further adoption of distributed

[4] http://owllink-owlapi.sourceforge.net/download.html

and/or scalable solutions, especially for the knowledge management components, as is the case in the following section.

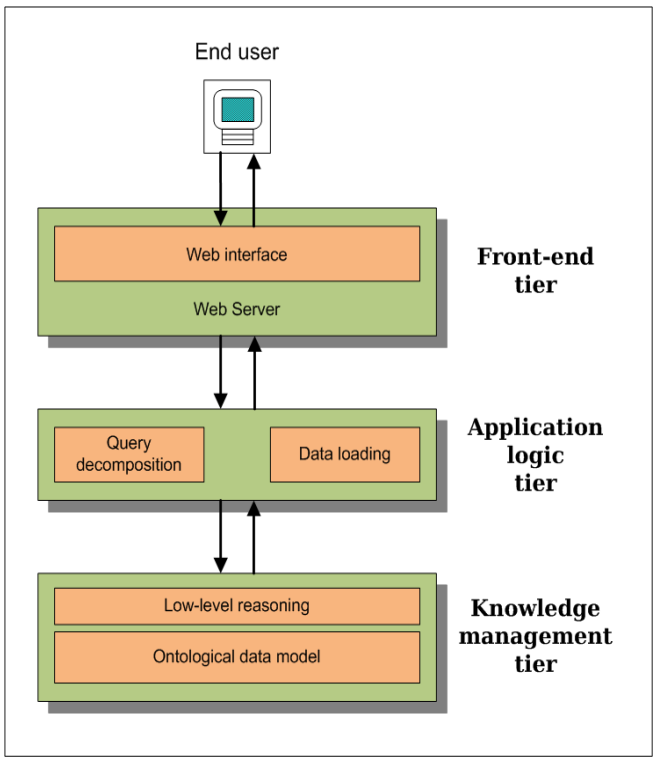

FIG. 1 3-TIER ARCHITECTURE FOR SEMANTIC WEB APPLICATIONS

Prototype Implementation

In order not to restrain an application to the particulars and/or limitations of any single inference engine and to allow for decentralized architecture at the same time, we have designed and developed a variant of OWL API (v. 2) that, in addition to its original advantages, can also support remote communication. As a result, this approach can accommodate any reasoner compliant with the OWL API, for example, FaCT++ or Pellet. In terms of Fig. 1 our middleware mediates between the knowledge management and the application logic tiers, with its server and client components, respectively. In the following, this approach is described and its validity is checked by investigating how it may contribute to the decentralized reasoning problem.

General Description

Our goal is to have one main server that hosts the inference engine, which would be able to provide reasoning services to a number of client applications over the TCP/IP protocol. Client applications will be able to make queries and receive responses from the knowledge base, exactly as if they were using the original OWL API.

To this extent, we developed a client-server interface in Java, using *remote objects*. The reasoner is hosted in a server along with the OWL API. Our *server* application listens to a specific port, waiting for connections. Our *client* application, a variant of the OWL API, is physically located on a different computer system and, in order to use the reasoner, it has to connect to the server. Upon connection, the server application initializes a `Reasoner` object, reads the parameters from the client application, calls the corresponding reasoner method, and sends back the results. The whole client-server communication and message transaction, take place over the TCP/IP protocol.

Server Application

In our server application (Fig. 2), all we had to do was to create a myServer class which, upon initialization, creates a TCP/IP ServerSocket on a desired port and awaits for client connections. Upon connection we call the WorkerRunnable() method, which initializes an OWLOntologyManager object and a Reasoner object, and accepts the client parameters. These parameters are deserialized with the readObject() command. After processing the desired client requests, the server sends back the results making use of the writeObject() method.

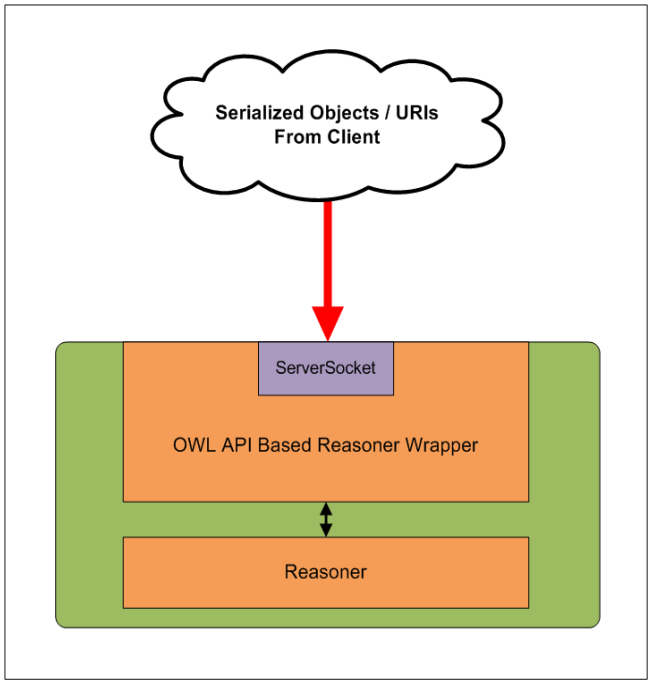

FIG. 2 SERVER APPLICATION

Multi-threaded Approach

In order to further extend the distributed characteristics of our middleware, we have also

developed another version of our server application that makes use of Java threads (Fig. 3). In this multi-threaded version, the main difference is that, instead of calling the WorkerRunnable() method, we create a new thread on the WorkerRunnable class. Thus, each client connected to our server is served by a different instance of WorkerRunnable, meaning a different instance of the Reasoner class.

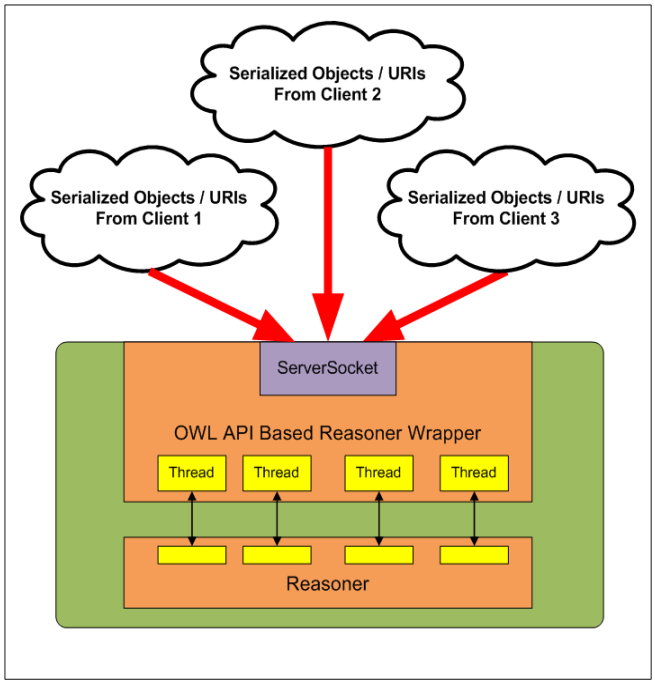

FIG. 3 MULTI-THREADED SERVER

Therefore, clients' requests are now processed in parallel, rather than in a serial client-after-client way. The only catch in the multi-threaded approach is that it cannot function with the FaCT++ reasoner, because of the JNI, which does not support multi-threading.

Client Application

In order to have a functional client application (Fig. 4), we have created a myReasoner class. Its constructor has three arguments: an OWLOntologyManager object, server's IP address, and server's port. When a myReasoner object is constructed, it creates a TCP/IP Java Socket, along with the corresponding ObjectOutputStream and ObjectInputStream that are required for the client-server communication. In addition, the myReasoner class contains all the reasoner-related methods defined in the original OWL API interfaces. These methods are replicate copies of the originals, in order to provide an interface that can be used instead of the original OWL API, with minimal code modifications. Thus, a standard OWL API application just needs to import our myReasoner class, instead of the original one (Table 2). From now on any method can be used, exactly as in the original OWL/API, i.e., reasoner.classify().

TABLE 2 EXAMPLE USAGE OF THE myReasoner CLASS

```
public boolean hasObjectPropertyRelationship
(OWLIndividual individual1,
OWLObjectPropertyExpression expression,
OWLIndividual individual2) throws IOException
{
out.writeInt(hasObjectPropertyRelationship);
 out.writeObject(individual1);
 out.writeObject(expression);
 out.writeObject(individual2);
 out.flush();
 boolean result = in.readBoolean();
 return result;
}
```

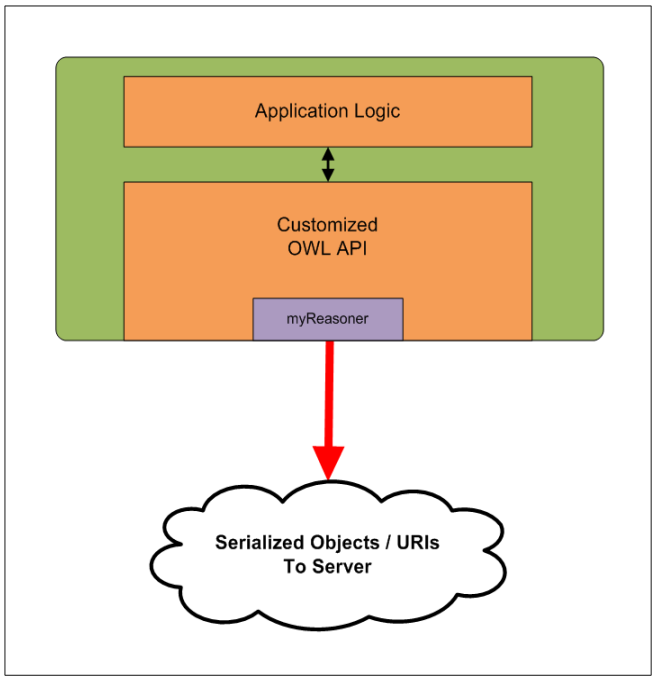

FIG. 4 CLIENT APPLICATION

TABLE 3 AN INDICATIVE METHOD OF THE MYREASONER CLASS

```
public boolean hasObjectPropertyRelationship
(OWLIndividual individual1,
OWLObjectPropertyExpression expression,
OWLIndividual individual2) throws IOException
{
out.writeInt(hasObjectPropertyRelationship);
 URI uri1 = individual1.getURI();
 out.writeObject(uri1);
 out.writeObject(expression);
 URI uri2 = individual2.getURI();
 out.writeObject(uri2);
 out.flush();
 boolean result = in.readBoolean();
 return result;
}
```

The parameters of these methods are either primitive Java objects (i.e., String, Boolean etc) or OWL API objects (i.e., OWLClass, OWLIndividual, OWLObjectProperty etc). In order to make it possible for our methods to send these parameters over TCP/IP, we utilize Java Serialization. We use the writeObject() and readObject() methods to convert a Java object into bytes and send it to the server through the TCP/IP Socket and vice-versa. An example of such a method in myReasoner is shown in Table 3.

URI-based Approach

By using the Java Serialization protocol in our implementation, it was observed that transferring Java objects, in byte format, tends to add a lot of communication load, because of the complicate structure of the OWL API objects. Every instance of an OWL API class inherits from many different classes and interfaces, resulting in the enlargement of the respective objects.

TABLE 4 AN INDICATIVE METHOD USING URIS

```
public boolean hasObjectPropertyRelationship
(OWLIndividual individual1,
OWLObjectPropertyExpression expression,
OWLIndividual individual2) throws IOException
{
out.writeInt(hasObjectPropertyRelationship);
out.writeObject(individual1);
out.writeObject(expression);
out.writeObject(individual2);
out.flush();
boolean result = in.readBoolean();
return result;
}
```

In order to overcome this drawback, we altered our implementation so that the OWL API objects, instead of being sent selfsame using Java Serialization, are replaced by their URIs whenever possible, such as in the OWLClass, OWLIndividual and OWLDataProperty classes. Thus we succeeded in minimizing the client-server communication load in many occasions. As an example, Table 4 shows how the myReasoner method in Table 3 can be transformed using URI passing.

The application has been tested with two inference engines that support the OWL API, namely FaCT++ (v. 1.3) and Pellet (2.0), showing some encouraging results, even in the case of very simple ontologies.

Results

In order to figure out our implementation's potential, we made some tests and time measurements based on the well known Example 8 of OWL API's documentation, using either the original pizza.owl [5] ontology or the Finance.owl one[6]. Both ontologies fall into the OWL DL dialect. We used two different computer systems: a client based on an Intel Atom 1.6 GHz processor with 2 GB of RAM, and a server based on an Intel Core 2 Duo 2.66 GHz with 2 GB of RAM.

During some indicative experiments, we measured the time (in *ms*) consumed by some methods as well as the total reasoning time in each case. *Total reasoning time* refers to the time the whole example program would take to terminate. Other metrics refer to various reasoning tasks and contribute to the total reasoning time, with classify() being the hardest from a complexity point of view.

Infrastructure Effect

To begin with, we tested the performance of FaCT++ and Pellet, running locally either on the server or on the client machine, as well as the effect of the hosting computer system. The results are summarized in Table 5.

It can be seen that, in the case of using FaCT++ in a local installation, the only time-consuming reasoner method is classify(). On the contrary, when using Pellet, we observed a considerable increase in total reasoning time compared to FaCT++, due to their different implementation strategies as well as Pellet's JVM dependency. Therefore we decided to stick to FaCT++ as the basis for our next tests.

In addition, it is shown that total reasoning time is related with the size and complexity of the corresponding ontology. As the finance ontology is larger and more complex that the pizza one, it is anticipated that reasoning operations will take longer to be completed.

Finally, by comparing the first and last column, it is indicated that the reasoning process is highly dependent on the computer resources available, like processing power: A powerful back-end system can seriously accelerate reasoning performance.

Remote Reasoning Effect

Afterwards, we tested all the possible topological cases: *local installation* on the client computer (no remote communication) and *client-server installation*,

[5] http:// www.co-ode.org/ontologies/pizza/2007/02/12/pizza.owl

[6] http://www.fadyart.com/ontologies/data/Finance.owl

against each ontology. These results are summarized in Table 6.

TABLE 5 TIME MEASUREMENTS FOR VARIOUS REASONING TASKS WITH LOCAL REASONER INSTALLATION ON DIFFERENT INFRASTRUCTURES

	Server, FaCT++, pizza	Server, Pellet, pizza	Server, FaCT++, Finance	Server, Pellet, Finance	Client, FaCT++, pizza
Total reasoning time	27	991	515	3082	239
Load Ontologies	0	30	0	966	0
classify	25	452	492	297	219
isConsistent	0	34	0	1106	0
Get Inconsistent Classes	0	29	0	31	0
Get escedantClasses	0	428	0	68	0

TABLE 6 TIME MEASUREMENTS FOR VARIOUS REASONING TASKS *WITH* AND *WITHOUT* REMOTE REASONER COMMUNICATION

	local, pizza	client-server, pizza	local, Finance	client server, Finance
Total reasoning time	*239*	*1866*	*2470*	*7460*
loadOntologies	0	636	0	5760
classify	219	195	2417	613
isConsistent	0	45	0	85
getInconsistent Classes	0	300	0	204
getDescedant Classes	0	193	0	242

It is observed that there is a significant time overhead on almost all occasions because of the remote communication over TCP/IP protocol and Java Serialization, as expected. On the other hand, using a "fast" server system, dedicated for reasoning processes, results in greatly boosting processor-demanding methods (such as classify()), thus decreasing the effect of remote communication delay. For example, the Finance ontology is classified almost 75% faster when this task is delegated to a more powerful system, through our middleware.

Bandwidth Effect

As a third step, we tried to calculate the effect of the network connection type (bandwidth) between the client and the server computer systems. Our implementation was tested using a fiber cable and a DSL line connection (2 Mbps download/256 Kbps upload), as shown in Table 7.

The results have confirmed that the total reasoning time of our implementation, as well as of any distributed application, highly depends on the type of network connection between the client and server computers, especially for the loadOntologies() method. It can be observed that in the case of sizeable ontologies, a very fast network connection can give great time results, theoretically verging on the local installation case.

TABLE 7 TIME MEASUREMENTS FOR VARIOUS REASONING TASKS ON DIFFERENT NETWORK CONNECTION TYPES

	pizza, fiber	pizza, dsl	Finance, fiber	Finance, dsl
Total reasoning time	1866	2214	7460	67362
loadOntologies	636	672	5760	65460
classify	195	252	613	644
isConsistent	45	85	85	123
getInconsistent Classes	300	332	204	316
getDescedant Classes	193	240	242	321

Discussion

An immediate conclusion of all the above is that the harder the ontology is, the better our middleware performs, as the relevant overhead lessens. And in real-life scenarios, ontologies tend also to evolve over time, thus classification results cannot always be cached. Therefore, the gains of using a dedicated reasoning server will ultimately overcome any communication overhead, making possible the use of "slow" computer systems for hosting the non-reasoner portions of any ontology-based semantic application.

TABLE 8 TIME MEASUREMENTS FOR VARIOUS REASONING TASKS WITH REMOTE REASONER COMMUNICATION USING SERIALIZED OBJECTS AND URIS

	local, pizza	client-server (Serialize), pizza	client-server (URI), pizza
Total reasoning time	239	1866	1758
loadOntologies	0	636	640
classify	219	195	196
isConsistent	0	45	42
getInconsistentClasses	0	300	215
getDescedantClasses	0	193	205

In addition, the Java Serialization protocol seems to contribute greatly to this communication load blow-up, due to the complicate structure of OWL API objects that have to be transferred. To this end, we have also experimented with our implementation so as to communicate only the objects' URIs, instead of sending them selfsame through Java Serialization. In this way,

the client-server communication load can be reduced in many occasions, even in the simplest case of the pizza ontology, as shown in Table 8.

Future work

In the future, we intend to further evaluate our middleware, especially in terms of its multi-threaded and load-balancing capabilities and test it with more state-of-the-art reasoners, like HermiT. The thread-safety of these reasoners is also a matter that requires further investigation.

Our minimal intervention with the OWL API ensures that it would be straightforward to align with its latest version 3. In addition, it would be interesting to compare our implementation with the OWLlink API, in terms of overall performance and communication overhead, and also examine how the latter can extend towards the multi-threaded scenario.

Conclusions

The realization of the Semantic Web imperatively calls upon ways to transfer its benefits to the end users of the World Wide Web. For this to happen, the facility for reasoning-based services should be made available by applications that would transparently integrate into the everyday browsing experience, in accordance to the Semantic Web vision.

It has been argued that an important step for Semantic Web applications is their advancement to distributed architectures, preferably 3-tier ones, because of their advantages in decentralized and load-balancing implementations. However, such a truly decentralized architecture, in accordance to the traditional 3-tier paradigm, has not been, until recently, possible with the majority of the current state-of-the-art and highly expressive inference engines. Possible reasons include limitations of their interface capabilities as well as incompetency of related protocols.

In this paper we have introduced a middleware that can support remote communication with any reasoner compliant with the OWL API, and shown that it is even possible to concurrently employ a reasoner for different requests, resulting in the actual parallelization of reasoning tasks. The performance gain of using high-end servers for these tasks is obvious. The effect of the communication overhead imposed degrades for complex and evolving ontologies and can be alleviated by following less bloated approach of URI passing.

Although the OWL API provides a sound framework for developing ontology-intensive applications, the decentralized deployment of reasoning services is not straightforward. The standardization of a protocol for this process, that would take into account the whole of OWL 2 as well as the full set of reasoner capabilities, even for complex query answering, is currently an active field of research and development and is desirable in any production system. The enrichment of the OWLlink specification and the development of the OWLlink API are considered to be very promising steps towards this direction.

REFERENCES

Berners, Lee T., Hendler, J. and Lassila, O.: The Semantic Web. Scientific American 279: 34-43.

Bock, J., Tserendorj, Y., Xu, Y., Wissmann, J. and Grimm, S. (2009): A Reasoning Broker Framework for OWL. In Proc. of the 5th Int. Workshop on OWL: Experiences and Directions.

Deshpande, Y. and Hansen, S. (2001): Web engineering: Creating a discipline among disciplines. IEEE Multimedia, April-June: 82-87.

Dickinson, I. (2004): Implementation experience with the DIG 1.1 specification, Hewlett Packard, Digital Media Sys. Labs, Bristol, Tech. Rep. HPL-2004-85.

Grau, B. C., Horrocks, I., Motik B., Parsia, B., Patel-Schneider, P. and Sattler, U. (2008): OWL 2: The next step for OWL. Web Semantics: Science, Services and Agents on the World Wide Web 6 (2008): 309-322.

Hendler, J. (2008): Web 3.0: Chicken Farms on the Semantic Web. Computer 41: 106-108.

Horridge, M., Bechhofer, S. and Noppens, O. (2007): Igniting the OWL 1.1 Touch Paper: The OWL API. In Proc. of the 4th Int. Workshop on OWL: Experiences and Directions.

Kazakov, Y. (2008): SRIQ and SROIQ are Harder than SHOIQ. In Proc. of the 21st Int. Workshop on Description Logics.

Koutsomitropoulos, D.A., Fragakis, M. and Papatheodorou, T.S. (2006): Discovering Knowledge in Web Ontologies: A Methodology and Prototype Implementation. In Proc. of SEMANTICS 2006 International Conference: 151-164.

Koutsomitropoulos, D.A., Solomou, G., Pomonis, T., Aggelopoulos, P. and Papatheodoroui, T.S. (2010): Developing Distributed Reasoning-Based Applications for the Semantic Web. In Proc. of the IEEE International Symposium on Mining and Web. WAINA: 593-598.

Liebig T., Luther, M., Noppens, O., Rodriguez, M., Calvanese, D., Wessel, M., Horridge, M., Bechhofer, S., Tsarkov, D. and Sirin, E. (2008): OWLlink: DIG for OWL 2. In Proc. of OWL Experiences and Directions.

Murugesan, S., Deshpande, Y., Hansen, S., and Ginige, A. (1999): Web Engineering: A New Discipline for Development of Web-based Systems. In Proc. of 1st ICSE Workshop on Web Engineering, International Conference on Software Engineering. Los Angeles, CA, USA.

Noppens, O., Luther, M. and Liebig, T. (2010): The OWLlink API: Teaching OWL Components a Common Protocol. In Proc. of the 7th Int. Workshop on OWL: Experiences and Directions.

Pomonis, T., Koutsomitropoulos, D. A., Christodoulou, S. P., and Papatheodorou, T. S. (2009): Towards Web 3.0: A unifying architecture for next generation web applications. In Handbook of Research on Web 2.0, 3.0 and X.0: Technologies, Business and Social Applications: 192-204. MURUGESAN S. (ed). IGI Global.

Rodriguez, M. and Calvanese, D. (2008): Towards an open framework for Ontology Based Data Access with Protégé and DIG 1.1. In Proc. of the 5th Int. Workshop on OWL: Experiences and Directions.

Tobies, S. (2001): Complexity results and practical algorithms for logics in Knowledge Representation. Ph.D. Thesis. RWTH-Aachen University.

Tzanetos Pomonis received his B.Sc. from Computer Engineering and Informatics Department in 2003, and M.Sc. in Computational Mathematics & Informatics in Education from Mathematics Department in 2007. He also received a PhD from the University of Patras in 2011. His research interests include Web Engineering, Web 3.0, Web Information Systems, Knowledge Management in the Web, Web 2.0, Artificial Intelligence in the Web, and the Semantic Web.

Dimitrios Koutsomitropoulos is a researcher at the High Performance Information Systems Laboratory (HPCLab), University of Patras. He has received a M.Sc. and a Computer and Informatics Engineer diploma, from the Computer Engineering and Informatics Department. He also received a PhD from the University of Patras in 2008. His research interests include knowledge discovery, automated reasoning, ontological engineering, metadata integration, semantic interoperability and the semantic web.

Georgia Solomou is a researcher at the High Performance Information Systems Laboratory (HPCLab), University of Patras. She has received a M.Sc. and a Computer and Informatics Engineer diploma, from the Computer Engineering and Informatics Department.

Panagiotis Aggelopoulos is a researcher at the High Performance Information Systems Laboratory (HPCLab), University of Patras. He has received a M.Sc. and a Computer and Informatics Engineer diploma, from the Computer Engineering and Informatics Department.

Sotiris P. Christodoulou received BSc in computer engineering and informatics in 1994, and PhD from the University of Patras in 2004. He is currently a senior researcher of Web engineering at HPCLab at the University of Patras. His research interests include Web Engineering, Hypermedia, Web Information Systems Development, XML.

Andreas B. Gizas is a researcher at the High Performance Information Systems Laboratory (HPCLab), University of Patras. He has received M.Sc. and a Computer and Informatics Engineer diploma, from the Computer Engineering and Informatics Department. He is currently a PhD candidate of the University of Patras. His research interests include Web Engineering, Hypermedia, Web Information Systems Development, XML and Web 2.0.